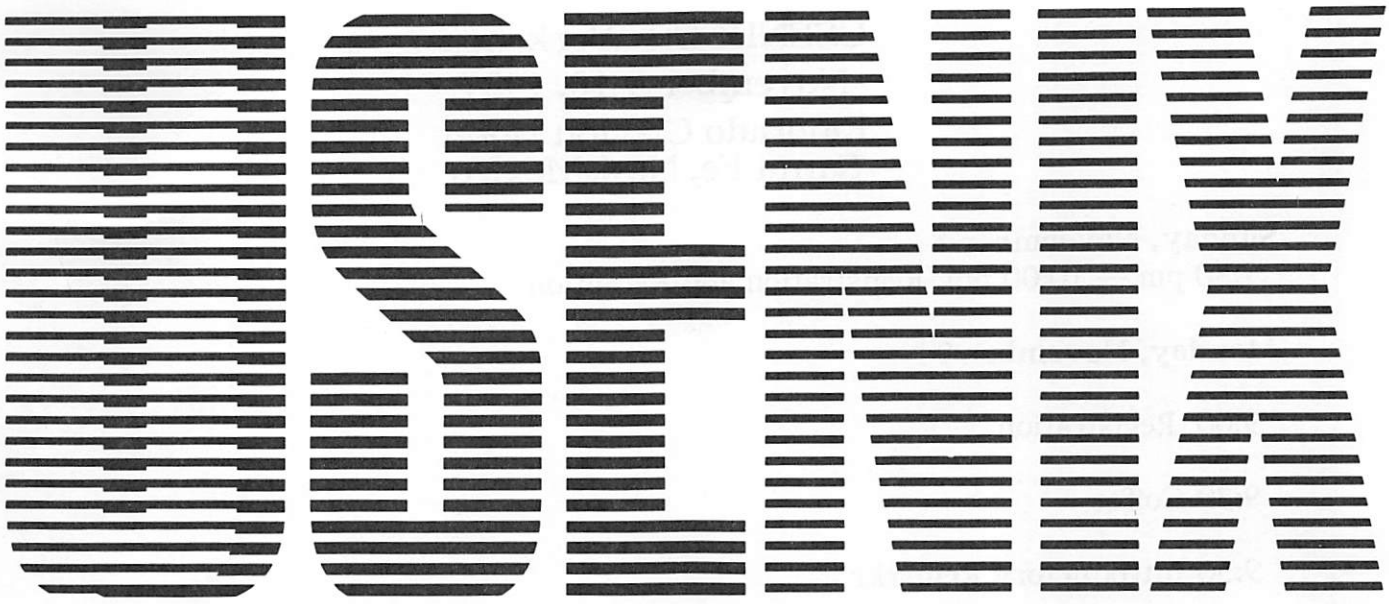


ABSTRACTS

C++ Workshop

Santa Fe, NM

November 9–10, 1987

Program

USENIX C++ Workshop
November 9-10, 1987
Eldorado Clarion Hotel
Santa Fe, New Mexico

Sunday, Novermber 8

7:30 pm — 10:00 pm Registration and Reception

Monday, November 9

9:00 Registration

9:30 Coffee

9:50 Introductory Remarks

10:00 Bjarne Stroustrup, AT&T Bell Labs: *The Evolution of C++ 1985 — 1987*

11:00 Steve Dewhurst, AT&T: *The Architecture of a C++ Compiler*

11:30 Michael Ball, TauMetric Corporation: *The Oregon Software C++ Compiler*

12:00 Lunch Break

1:30 John Carolan, Glockenspiel: *C++ for OS/2*

2:00 Ken Friedenbach, Apple Computer: *Porting C++ to the Macintosh OS*

2:30 Philippe Gautron and Marc Shapiro, INRIA: *Two Extensions to C++: A Dynamic Link Editor and Inner Data*

3:00 Jonathan Shopiro, AT&T Bell Labs: *Extending the C++ Task System for Real-Time Control*

3:30 Coffee Break

3:45 Tom Doeppner and Alan Gebele, Brown University: *C++ on a Parallel Machine*

4:15 Roy Campbell, University of Illinois at Urbana-Champaign: *CHOICES — A Multiprocessor Object-Oriented Operating System*

4:45 Oliver McBryan, New York University: *C++ Environments for Parallel Computing*

5:15 Dave Detlefs, Carnegie-Mellon University: *Avalon: C++ Extensions for Transaction-Based Programming*

5:45 Dinner Break

8:00 Short Discussions

Steve Mahaney and Ravi Sethi, AT&T Bell Labs: *Concurrency and C++*

John Rose, Thinking Machines, Inc.: *C*: A C++-like Language for Data-parallel Computation*

Kevin Kenny, University of Illinois at Urbana-Champaign: *Encapsulators — A Metaphor for Software Organization in C++*

Peter Kirslis, AT&T: *A Style for Writing C++ Classes*

Kenneth Brown, Cardinal Information Systems, Inc.: *Object-Oriented Databases*

Judith Grass, AT&T: *Using C++ in Language Processors*

James Coggins, University of North Carolina: *A Summary of C++ Applications Under Development at the University of North Carolina*

Michael Tiemann, MCC: *G++: A Free C++*

Dan Schuh, University of Wisconsin: *Parameterized Types for C++*

Mark Rafter, Warwick University: *Extending Stream I/O to include Formats*

9:45 Open Discussion

Tuesday, November 10

9:00 Coffee

9:30 Bjarne Stroustrup, AT&T Bell Labs: *What is 'Object-Oriented Programming'*

10:30 Keith Gorlen, National Institutes of Health: *OOPS: A C++ Object-Oriented Program Support Class Library*

11:00 Coffee Break

11:15 Ken Fuhrman, Ampex Corporation: *An Object-Oriented Class Library for C++*

11:45 Tsvi Bar-David, AT&T: *Teaching C++*

12:15 Lunch Break

1:45 Al Conrad, University of California at Santa Cruz: *Modelling Graphical Data with C++*

2:15 James Coggins, University of North Carolina: *Integrated Class Structures for Image Pattern Recognition and Computer Graphics*

2:45 Jim Waldo, Apollo Computer Inc.: *Using C++ to Develop a WYSIWYG Hypertext Toolkit*

3:15 Coffee Break

3:30 Mark Linton and Paul Calder, Stanford University: *The Design and Implementation of InterViews*

4:00 Mark Linton, Stanford University: *The Design of the Allegro Programming Environment*

4:30 Ragu Raghavan, Mentor Graphics: *A C++ Class Browser*

5:00 Tom Cargill, AT&T Bell Labs: *Pi: A Case Study in Object-Oriented Programming*

5:30 Adjournment

Conference Chair:

Keith Gorlen
Building 12A, Room 2017
National Institutes of Health
Bethesda, MD 20892

301-496-5363 usenix!nih-csl!keith

USENIX Conference Coordinator:

Judy DesHarnais

USENIX Conference Liasion:

David A. Yost

Preprint Production:

Tom Strong

The Evolution of C++: 1985 to 1987

Bjarne Stroustrup

ABSTRACT

The C++ programming language describes C++ as defined and implemented in August 1985. This paper describes the growth of the language since then and clarifies a few points in the definition of the language. It is emphasized that these language modifications are extensions; C++ has been an will remain a stable language suitable for long term software development. The main new features of C++ are: multiple inheritance, recursive definition of assignment and initialization, class specific new and delete operators, protected members, overloading of operator -> and pointers to members. Finally, a few thoughts about the likely direction of further evolution of the language are presented.

Introduction

As promised in *The C++ programming language* C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a very great range of application ares. The primary aim of the extensions has been to enhance C++ as a language for data abstraction and object-oriented programming in general and to enhance is as a tool for writing high quality libraries of user-defined types in particular. All of the features described here are in use within AT&T and the C++ implementation that supports them will become generally available within months.

A programming language is only one part of a programmers world. Naturally, work is being done in many other fields (such as tools, environments, libraries, education and design methods) to make C++ programming more pleasant and effective. However, this paper deals strictly with language and language implementation issues.

1 Multiple Inheritance

A class may be derived from more than one direct base class:

```
struct A { f(); };
struct B { f(); };
struct C : A, B {};
```

Ambiguous uses are detected (at compile time):

```
void g() {
        C* p;
        p->f();  // error: ambiguous
}
```

Typically one would resolve the ambiguity by adding a function:

```
class C {
        void f() {
                // do C's own stuff
                A::f()
                B::f();
        }
}
```

A class can appear more than once in an inheritance DAG:

```
class A : public L { ... };
class B : public L { ... };
class C : public A, public B { ... };
```

In this case C has two sub-objects of class L: A::L and B::L.

Virtual base classes provide a mechanism for sharing between and expressing dependencies among sub-objects in an inheritance DAG:

```
class A : public virtual W { ... };
class B : public virtual W { ... };
class C : public A, public B, public virtual W { ... };
```

In this case there is a single (shared) sub_object of class W.

2 Recursive Definition of Assignment

Assignment and initialization is default defined as memberwise assignment and memberwise initialization:

```
struct A { ... void operator=(A&); A(A&); };

struct B { int x; A a; char* p; };

void f() {
        B b1;
        B b2 = b1;          // B::a is initialized using A(A&);
        b1 = b2; // B::a is copied using A::operator=(A&);
}
```

3 protected Members

A class member can be declared protected:

```
class node {
   // private stuff
protected:
   node* left;
   node* right;
   // more protected stuff
public:
   virtual void print();
   // more public stuff
};
```

a protected member is private as far as functions outside the class hierarchy are concerned but accessible to member functions of a derived class in the same way that it is accessible to members of its own class:

```
class my_node : public node {
   void f() { left = 0; }   // OK: my_node::f is member of a class
                            //     derived from node
};

void f() { left = 0; }      // error: f is global and left is protected
```

4 Pointers to Members

Pointers to class members can be defined and used:

```
struct s { int f(char*); }

int (s::* pmf)(char*) = &s::f;
```

This defines pmf to be a pointer to a member of class s of type function taking a char* argument and returning an int; pmf is initialized to point to s::f. To call a member function through a pointer an object of the right type must be supplied (in addition to whatever arguments the function require):

```
void g()
{
        s obj;
        s* ptr = &obj;
        int i = (obj.*pmf)("call1");  // call through pmf for obj
        int j = (ptr->*pmf)("call2"); // call through pmf for *ptr
}
```

Pointers to data members and pointers to virtual functions are allowed.

5 Class Specific Operator new

X::operator new() and X::operator delete() can be defined for a class X. This allows a user to allocate space for objects of class X and its derived classes using a special allocator without assigning to this:

```
class X {
        // ...
        void* operator new(long s) { return my_alloc(s); }
        void operator delete(X* p) { my_free(p); }
};
```

X::operator new() will be usedto allocate space for objects of class X instead of the global operator new().

6 Overloading Operator ->

Operator -> can be overloaded:

```
struct Y { int m; };

class X {
        Y* p;
        Y* operator->() { return p; };
};

void f() {
        X a;
        a->m;   // becomes a.p->m
}
```

This makes it easier to create classes of objects intended to act as "smart pointers" and much more convinient to use such objects.

The Architecture of a C++ Compiler

Stephen C. Dewhurst, AT&T

This presentation covers several aspects of project and design goals applied to development of a C++ compiler in C++ and the results obtained. The work described was undertaken at AT&T jointly by Laura Eaves, Kathy Stark and the author.

The first section, designed to elicit sympathy for compiler writers in general and the author in particular, details the various considerations—theoretical, technical, and organizational—that shape a compiler design. These include considerations for efficiency and portability, compatibility with existing translators, code and perceived language standards, corporate product goals, and the developers' *own* goals.

An overview of the structure of the compiler shows how the design reflects these goals, and serves as background for the remaining sections. This design shows clearly how C++ supports the effective melding of multiple programming paradigms: Subsections of the compiler are reflective of procedural, data abstraction, and object-oriented modes of programming, as well as hybrids of these.

The bulk of the presentation addresses two important properties of the compiler: It's extreme portability and its ability to function within multiple compilation paradigms. Because we expect C++ to be widely used, we did not want to restrict the range of applicability of the compiler to a small set of machines or operating systems. In making the (relatively safe) assumption in the design of the compiler intermediate representation that we are translating C++, and in defining targets as code generators rather than machines, we are able to achieve portability without concommitant compile-time performance problems. By the same token, we have minimized the linkage of the compiler design to assumptions about the model of compilation under which it is run. Current capabilities include two or one-pass compilation and single-process, multiple file compilation. Future work within the same framework may allow stacking and saving of compilation states to avoid recompilation, precompiled program pieces, and object libraries.

A final section offers gratuitous advice in presenting a number of general principles of compiler design gleaned from this project.

The Oregon Software C++ Compiler

Michael S. Ball
TauMetric Corporation

1. General Description

The Oregon Software C++ compiler is based on its existing Pascal-2 compiler, and utilizes the optimizer and code generators from that compiler. This provides a mature base on which to build plus a variety of target machines available with a minimum of work. The compiler is primarily a C++ compiler, but compiler switches allow it to function as an ANSI C compiler. The result is an optimizing C++ compiler which directly translates C++ into object code for the target system.

There is currently no standard for the C++ language, and the Stroustrup book is not complete enough to serve that purpose. This may be inherent in a new language still undergoing change, but it makes the compiler writer's job much more difficult. We chose to use the book as the first guide, followed by the results of executing the current C++ frontend (we don't have a source license), then personal comments from Stroustrup. We assume that we will have to make changes as better data becomes available from AT&T.

We chose Keith Gorlan's OOPS package as the basic validation test, since it is a relatively large body of code which has seen wide dissemination.

The initial implementation is on a Sun 3 workstation, and is currently being tested.

2. Advantages of a Compiler

The major advantage of compiling C++ directly, instead of through the target C compiler, is improved system integration and debugger handling. For example, providing direct access to the C++ symbol table allows the debugger to deal directly in C++ terms. Dealing with concepts such as overloading and member variables is much simpler when the debugger has direct access to the C++ symbol table.

The other advantage is the removal yet another pass from the compiler. Compiling into C code has obvious advantages in portability, but it does sacrifice some performance and convenience.

3. Implementation

The modifications to the compiler consist of a new lexical scanner, including an ANSI standard preprocessor, and a new syntactic and semantic analysis pass. The output of the analysis stage is a language-independent intermediate language which is then processed by a slightly modified version of the Pascal-2 optimizer and code generator.

The new compiler was designed originally as a C++ compiler with internal mechanisms for handling all C++ constructs. Compiler switches were provided to process ANSI and earlier versions of C. The ANSI version was checked out first, primarily because it is an (almost) subset of C++ and because there are validation and test suites available for it. After this version passed the validation suite, C++ features were enabled one at a time and checked out incrementally before testing the compiler as a whole.

The whole point of implementing a compiler for C++ is to increase the ease with which the user can debug his code. To ease this task, there are compiler switches to enable runtime checking code for common errors such as a reference through a NULL pointer. An object oriented debugger which is expected to be completed before the product is released works with this checking code to speed error detection and elimination.

C++ for OS/2

John Carolan
Glockenspiel Ltd.

Abstract

MS-DOS with Windows 2.0 and OS/2 with Presentation Manager represent a vast opportunity for C++ in terms of programmer population.

C already dominates the language scene for these environments, so the sales prospects for a 'better C', which addresses the specific engineering problems of presentation oriented software, have got to exceed those for any earlier language.

The paper summarises the engineering problems in presentation systems which select in favour of C++.

Then it lists many issues which require urgent attention before C++ can be accepted by windows developers. These issues include technical ones, such as inferior run-time efficiency, lack of debugger support and lack of a verification technology for C++. They also include the present marketing chaos:

AT&T emphasise the Ada-competitive features of C++, the OOPS community emphasise the Smalltalk-competitive features, but what people want to evaluate on OS/2 are the C-competitive features!

The main portion of the paper describes solutions now under construction which address the technical problems that could hinder the OS/2 community's rapid assimilation of C++.

It concludes by listing the unique benefits of C++ for developing software whose architecture is dominated by human interface concerns — particularly under OS/2 and systems incorporating X Windows.

For C++ to succeed here, we must present it as

A mature language with unique benefits for human interface.

C++ on the Macintosh

Ken Friedenbach, Ph.D.
Development Systems Group
Apple Computer, Inc.
USENIX C++ Workshop
Nov. 9-10, 1987 Santa Fe, N.M.

1. **Introduction.** This talk discusses work in progress on a port of the AT&T CFront compiler [1] to the Macintosh Programmers Workshop (MPW) development system. [2] There are several objectives in providing C++ for the Macintosh. First, C++ will provide a standard object oriented language for C programmers developing applications for the Macintosh. Second, Apple will provide C++ interfaces to all the Macintosh operating system and toolbox functions. Third, the implementation will allow mixing Object Pascal and C++ in application programs. Finally, building on the ability to mix C++ and Object Pascal, Apple will provide C++ interfaces to MacApp, the extensible Macintosh Application.

C++ was chosen over other C-like object oriented languages because it is highly compatible with both C and ANSI C, the CFront compiler from AT&T has been tested in production use, and derived classes of C++ provided a close semantic match for interfacing with the methods, inheritance, and overriding of objects in Object Pascal.

The talk will give a brief description of the MPW development system, Object Pascal, [3,4] and MacApp.[5] Several examples of Object Pascal code, and fragments of MacApp source code will be discussed.

2. **Apple Extensions to C++.** There are three areas where Apple has extended C++ for the Macintosh: numerics, Pascal calling conventions, and support for Object Pascal.

Numerics. The Standard Apple Numerics Environment (SANE) is an IEEE-754 compatible environment that Apple has supported on all of its machines. Support of SANE implies adding some new numeric types to C++: `extended` (an 80-bit or 96-bit floating point type), and `comp` (a 64-bit integer type). Both ANSI C and C++ still prefer double precision, and do not allow extended precision everywhere, e.g. in arguments and function return values in libraries. At present, Apple is using extended precision in libraries (including `stream`, `in`, and `out`) for both speed and accuracy.

Pascal conventions. For historical reasons, the Pascal run-time conventions on the Macintosh differ from the standard C conventions. In order to support these differences efficiently, and still allow mixing of Pascal and C code, the languages have been extended to indicate external definitions in a different source language.

In the present MPW C compiler, a Pascal procedure might be declared as follows:

```
void pascal SomeProc (short x, window * wp)
    extern;
pascal void DrawText (Ptr textBuf, short firstByte,
    short byteCount)  extern 0xA885;
```

Some support has been added to CFront for this use of `pascal` as a storage class, for the use of `void` as a return type, and for directly calling Macintosh operating system and toolbox functions with A-Traps.

Support for Object Pascal. In order to support Object Pascal two new keywords, `indirect` and `inherited`, have been introduced. Indirect might be viewed (especially for compatibility) as a predefined, empty base struct:

```
struct indirect { };
```

But in the Macintosh implementation of objects, indirect has special semantics. It is used to indicate that the structure or class will be named, allocated, and accessed in the same manner as objects in Object Pascal. Virtual member functions of indirect classes are called via the method dispatch techniques of Object Pascal. Indirect objects are allocated on the heap and accessed via a handle (a pointer to a pointer). Naming conventions for indirect virtual classes match Object Pascal and trigger optimization by the linker.

The word keyword `virtual`, like the keyword `indirect`, can be viewed as a pragma to a strong compiler which is compensating for a weak linker. Both may disappear when the back-end C compilers support ANSI C, and the standard system linkers become more intelligent about doing global analysis and optimizing run-time procedure tables.

3. **Comparison of C++ and Object Pascal.** Some conclusions can be drawn about C++ and Object Pascal as programming languages.

Classes in C++ are more orthogonal than objects in Pascal. In C++ functions are allowed as members of structs or classes. Like normal structs, classes can be global variables, automatic variables, or elements of an array. In Object Pascal, objects can only exist individually on the heap and are accessed through a handle. This limits the efficiency of objects and their usefulness for problems which require more than a hundred or so objects. It also makes the mixing of normal Pascal code and objects more difficult than in C++.

Methods in Object Pascal are simpler and easier to use. In C++, a function must be declared as either virtual or not virtual (normal member) in the source code. In Object Pascal all methods are essentially virtual. But the linker performs optimization which reduces methods with single implementations to being normal procedures. This is important for large systems of software, since it preserves the ability of individual developers to override all standard implementations without paying a uniform run-time cost.

Hiding the details of Macintosh memory management is good. Currently, objects in Object Pascal and indirect classes in C++ are the only language constructs which are high enough to effectively hide the details of memory management from code for data access. In using the normal declarations for accessing Macintosh operating system and toolbox structures, code becomes quite specific in dereferencing pointers once and handles twice.

The Macintosh memory management scheme has been a good one, in terms of packing a lot of functionality into limited memory sizes. However, it is a radically different scheme, from traditional memory management, and it is likely to want to change when MMU's become widely available for personal computers with large memory sizes.

The cost of hiding the details of memory management is too high. Indirect classes in C++ should not be supported, because they are contrary to the intent of C. Hiding a level of indirection for something as simple as a data reference, can lull the C++ programmer into writing inefficient code. This is especially true for a processor such as the 68000 which has 32-bit addresses and only a 16-bit data bus. In following three levels of indirection (one for accessing the parameter `this`, and two for dereferencing a handle) the processor typically does at least nine memory references (three instructions and six data fetches).

In fact, the high cost of indirection was of great concern from the earliest days of Object Pascal (Classcal on the Lisa). Implicit references to SELF and the dereferencing of an object handle were carefully added to the register optimization of the Pascal compiler.

4. **The future.** Apple has defined a subset of C++, called "Minimal C++", [6] which it intends to use in implementing the interfaces to MacApp. Apple is encouraging developers of C compilers for the Macintosh to support minimal C++ as a standard way of providing object oriented features in C and ANSI C compatible compilers.

Apple would like to see improved compatibility between ANSI C and C++. But it is not always obvious which language is correct. Some areas of concern: the identifier name spaces could be made the the same; agreement about the status of tags in `struct` and `union` declarations; the `const` construct has a few problems, e.g. named expressions as array bounds; and function declarations with no arguments or a variable number of arguments differ.

Acknowledgements

David Goldsmith directed the investigation and design phases of the C++ project as part of the MacApp project. Cliff Greyson moved the CFront compiler to MPW and made the Apple extensions. Keithen Hayenga and Ron Metzger produced the C++ interfaces to the Macintosh operating system and toolbox. Clayton Lewis directed the numerics work for C++. Joe MacDougald is developing sample programs and a test suite for C++. Don Reed is preparing technical documentation.

References

1. "AT&T C++ Translator: Release Notes for version 1.2", AT&T Software Sales and Marketing, Greensboro, NC.
2. "MPW 2.0 Reference", Apple Programmers Development Association (APDA), Seattle, WA, 1987.
3. "MPW 2.0 Pascal Reference", APDA, Seattle, WA, 1987.
4. "Introduction to Object Pascal", Ken Doyle, MacTutor, MacTutor Company, Placentia, CA, Dec 1986. (reprinted by Advanced Technology Group(ATG), Apple Computer, Cupertino, CA.)
5. "MacApp Release 1.1.1", software and documentation, APDA, Seattle, WA, 1987
6. "Minimal C++ Report", David Goldsmith and Larry Tesler, Apple Technical Report, ATG, Cupertino, CA, Feb, 1987.

Two extensions to C++: A dynamic link editor and inner data

Philippe Gautron
Marc Shapiro

Institut National de Recherche en Informatique et Automatique
B.P. 105, 78153 Le Chesnay C'edex, France
uucp: gautron@corto.inria.fr

ABSTRACT

We present the design and implementation of a dynamic loader and linker for the C++ language. The motivations and goals of this tool are discussed, as well as its introduction into a C++ compilation chain. We also discuss "inner data", an alternative style of single inheritance, and a complement to dynamic linking. In each case, a keyword was added the C++ language, and a few modifications to the compiler were necessary. The implementation is clean and portable. We present the rationale of this work, the necessary changes to the language, and a working implementation.

This research is financed in part by the European Community, under Esprit contract SOMIW no. 367.

Extending the C++ Task System for Real-Time Control

Jonathan E. Shopiro

Bell Laboratories
Murray Hill, New Jersey 07974

EXTENDED ABSTRACT

The C++ task library is a coroutine[1] support system for C++. It was one of the first libraries written in C++ and it has served admirably in several applications. It is small, efficient, and easy to use. As part of a robot control project, it was extended to support real-time control, revised to take advantage of new features of C++ and to use more robust algorithms, and ported to the Motorola 68000 processor architecture. The new task system is compatible with the original.

A task is an object with an associated coroutine. The task library includes a scheduler that enables each task to execute just when it has work to do. In the C++ task system, a task does not have to wait when it initiates a coroutine, but instead can be made to wait when necessary for whatever is needed.

Programming with tasks is particularly appropriate for simulations, real-time process control, and other applications which are naturally represented as sets of concurrent activities. A task can represent a simple part of a complex system, and when the task gains control, it can process its current input data, perhaps creating other data that will be processed by other tasks. It can then relinquish control, waiting for more input or an external event.

In a program using the C++ task system, all tasks share the same address space so that pointers can be passed between tasks, and it is easy to share common data structures. Also, the scheduler is non-preemptive, so that each task runs until it explicitly gives up the single processor, and only then does the scheduler choose a new task to run. This eliminates the need for locks on shared data (which would be required if preemptive scheduling or multiple processors were used) and allows task-switching to be accomplished with low overhead, but requires the programmer to be careful that no task monopolizes the processor.

The rest of this section is an overview of control flow in the task system along with a brief note on task system performance and a description of the interrupt handler class and how it can be used to provide real-time response to external events. The full paper gives a detailed explanation of the internals of task switching and creation.

Control in the task system is based on a concept of *pending* objects. An object is pending if it has an operation that must wait. For example, a queue head is pending when the queue is empty, because the *get* operation must wait until there is something in the queue, and a queue tail is pending if the queue is full, because the *put* operation must wait until there is space in the queue. Each different derived class of object can have its own way of determining whether it is pending or not. When a task executes one of these operations on an object, if the object is not pending the operation succeeds immediately, but if it is pending, the task is said to be *blocked*, and must wait. The scheduler then chooses the next task from the *run chain*.

Each pending object contains a list (the *remember chain*) of the tasks that are waiting for it. When any operation changes the state of a pending object so that it is no longer pending, those tasks are moved to the run chain; this is called an *alert*. Thus the cycle is: a task runs until it blocks; it is

1. Coroutines can exchange control among themselves more freely than ordinary functions and procedures. In the usual function calling discipline, when one procedure (more precisely, one instance of a procedure) executes a procedure call, a new instance of the called procedure is created, and the calling procedure waits until the called procedure (and any procedures *it* may call) returns. A procedure instance is initiated when the procedure is called and is destroyed when it returns. When one coroutine (coroutine instance) initiates another it need not wait for the new coroutine to end, but instead it can be resumed while the new coroutine is still active. A running coroutine can relinquish control to any waiting coroutine without abandoning its state and later regain control and continue from where it left off.

saved on the remember chain of a pending object; some other task or an interrupt alerts the object; the original task is moved to the run chain; eventually the task runs again.

The fundamental operations of the task system are task creation and task switching. In order to make a meaningful evaluation of their performance, equivalent programs using tasks and UNIX† Operating System processes were written. These programs are given in the appendix of the full paper. The results were that task creation was 37 times faster than UNIX process creation, and task switching was 10 times faster than UNIX process switching.

It is important to note that the task system and the UNIX Operating System are not equivalent and that the results of these performance measurements do not imply that the task system is 23.5 times better than UNIX.

The application that motivated this work on the task system was a control system for two robots operating in the same workspace. The most important requirement of this application that was not fulfilled by the original task system was the need for tasks to wait for external events. A related requirement of some real time systems is to respond to external events in a timely manner, for example to retrieve data from an unbuffered external device. Also, in the original task system, the scheduler would exit when the run chain was empty. This is inappropriate in a system that is intended to respond to external events because some task might become runnable after an interrupt.

In the task system events that can be waited for are represented by instances of class object or derived classes. When the function object::alert() is called, the tasks that were waiting for that object are made runnable. A natural solution to the problem of waiting for external events was to define a new kind of object to represent external events, and when such an event occurs, to call object::alert() for the appropriate object. These objects are called interrupt handlers.

After an interrupt handler is created, a task can wait for it, exactly as for any other object. When the interrupt occurs, the handler's interrupt() function will be executed immediately. At the next entry to the scheduler, when the currently running task blocks, the interrupt handler will be alerted. Thus the waiting task becomes runnable. As long as any interrupt handler exists, the scheduler will wait for an interrupt, rather than exiting when the run chain is empty. An interrupt handler is always pending.

Interrupt_handler::interrupt() is a null function, but it is virtual, so that the programmer can specify the action to be taken at interrupt time by simply defining an interrupt() function in a class derived from Interrupt_handler. An example is given in the full paper. In this way real-time response can be obtained without resorting to a preemptive, priority-based scheduler which would be more complex and less efficient, and would require locking of shared data structures.

---

† UNIX is a registered trademark of AT&T.

C++ on a Parallel Machine

Summary

Thomas W. Doeppner Jr.
Alan J. Gebele[1]
Department of Computer Science
Brown University
Providence, RI 02912

Concurrent C++

Utilizing the advantages of parallel programming on a computer with multiple processors can often be difficult because of the lack of suitable programming language support for writing parallel programs. We have built a set of constructs developed for the language C++ which give the programmer a usable set of abstractions to write readable code using a parallel computer such as the Encore Multimax[2]. This tasking package was developed using a run-time system called Threads (described subsequently), which provides inexpensive support for concurrency on workstations and parallel processors. The implementation of the various classes of data abstractions in C++ is derived from the system described in [Stroustrup].

The tasking package incorporates several changes to facilitate true parallel processing instead of coroutine processing. It also adds a facility for the use of monitors for data access synchronization. The most important change is that the tasking package uses priority scheduling with the option to have preemptive scheduling on all tasks instead of the virtual time scheduling used by Stroustrup's tasks. The virtual time scheduling is useful for writing programs for simulation type applications but the tasking package gives a more general facility. By allowing the programmer to define more complex arrangements that allow more than one task executing at a time, true parallel processing can be achieved since the different tasks can be scheduled on more than one processor at a time. The tasking package also uses a different synchronization of tasks that uses wait queues associated with each task instead of the wait/alert method used in Stroustrup's classes. In addition to task to task synchronization, the tasking package also gives the programmer a class for defining derived classes as monitors. The use of monitors takes the place of synchronizing on classes derived from class *object* used in Stroustrup's Classes. The interface for declaring tasks, and manipulating them stays basically the same in the tasking package.

Threads

The Threads system cheaply supports the concept of a *thread of control* (or *thread*), which is an independent unit of execution, capable of concurrent execution with other threads. Our implementation is on top of workstations running Berkeley UNIX[3] (it currently runs on Suns, MicroVAXes and Apollos) as well as on a shared-memory multiprocessor — the Encore Multimax. It has proved to be fast enough to satisfy the needs of several projects at Brown and is about to be distributed to researchers at other institutions. The programming interface provided by Threads insulates the user from such details as the number of processors being used: programs written for uniprocessors normally work correctly on multiprocessors. We provide a standard set of high-level programming abstractions and also provide facilities for the programmer to create his or her own abstractions.

The first version of Threads was completed in September 1985, and versions have been used by researchers other than the implementer since then. The first multiprocessor version of Threads was completed in March of 1987, two months after we acquired our Encore. This version has been

---

[1] Gebele is currently affiliated with Bellcore and was supported by Bellcore for this work.

[2] Encore and Multimax are trademarks of the Encore Computer Corporation.

[3] UNIX is a trademark of AT&T Bell Laboratories.

used by others since April of 1987. A detailed tutorial on the use of the system is available [Doeppner 1]. [Doeppner 2] describes the design of the system.

The notion of cheap concurrency is often known as "lightweight processes." To our knowledge, this concept was first introduced as part of Xerox's Pilot system [Redell]. Other UNIX-based lightweight process implementations have been discussed in the past few years [Binding, Kepecs]. The system that comes closest to ours is Eric Cooper's C-Threads [Cooper]. Recently an operating system-supported notion of lightweight process, also known as a thread, has been implemented and used extensively at CMU as part of the Mach system [Tevanian].

What differentiates our system from all of the other UNIX-based systems is its complete support for systems concepts, including I/O, interrupts and exceptions. What differentiates our system from all of the other approaches to inexpensive concurrency is its support for concurrent programming abstractions — both the design of the particular abstractions we have implemented and how we allow the programmer to define new abstractions.

For a highly concurrent style of programming to be practical, threads, which support the concurrency, must be very inexpensive; the overhead required for creating, synchronizing and scheduling threads must be very low. A thread is not a traditional operating system process, which are typically very expensive to create and very expensive to synchronize. One reason for this expense is that processes are much more than just threads of control. They entail (usually) a separate address space and other protection boundaries which are time-consuming to set up. Another reason for the expense of processes is that they are managed by the operating system kernel; requests to perform operations such as synchronization must be passed to the kernel over a user-kernel boundary that is typically fairly expensive to cross (for example, for Berkeley UNIX running on a MicroVAX[4] II, the overhead for a trivial system call is 200 microseconds).

In the Mach system [Tevanian], threads are supported in the kernel, but appear to be cheap enough to qualify as lightweight processes (Mach threads are a lower-level abstraction than our threads). We are very interested in combining our approach with that of Mach, building our threads on top of Mach threads.

Currently all of our Threads system runs as user-mode code. This has resulted in a minimal overhead due to system calls and has allowed our system to be ported fairly easily to other UNIX systems.

Our implementation consists of two layers. The bottom layer, which is built on top of the UNIX process, implements the basic notion of a thread as an independent entity. A thread at this level presents a fairly low-level procedural interface which allows it to be manipulated directly. In the next layer, the thread abstraction is extended to supply the functionality needed to give the programmer the types of programming constructs expected in a high-level language; this layer can be thought of as the runtime library for such a language. A user of the Threads system may add additional layers, building on top of the lower layers by supplying additional procedures and adding additional fields to the thread data structures.

The functionality defined in the bottom layer includes scheduling and context switching, low-level synchronization, interrupt processing, exception handling and stack handling. In addition, this layer provides the routines employed for protection from interrupts and for the locking of data structures when used on a multiprocessor.

In the second layer we build up a set of programming constructs from the low-level interface of the bottom layer. These constructs allow threads to synchronize their execution (using semaphores or monitors [Hoare]), to perform I/O, and to respond to exceptions and interrupts. Exception handling is integrated with synchronization so that, for example, when a thread is forced out of a synchronization construct by an exception, the state of the synchronization construct is "cleaned up" so that it will continue to operate correctly. The programmer may choose a variety

[4] MicroVAX is a trademark of the Digital Equipment Corporation.

of ways of dealing with interrupts. For example, a thread may be created in response to an interrupt or an interrupt may cause an exception to occur in a specified thread.

References

[Binding] Binding, C., "Cheap Concurrency in C," *SIGPLAN Notices*, Vol. 20, No. 9, September 1985.

[Cooper] Cooper, E.C., Draves, R.P., "C Threads," Draft of Carnegie Mellon University/Computer Science Report, March 1987.

[Doeppner 1] Doeppner, T.W. Jr., "A Threads Tutorial," Computer Science Technical Report CS-87-06, Brown University, March 1987.

[Doeppner 2] Doeppner, T.W. Jr., "Threads — A System for the Support of Concurrent Programming," Submitted for publication, also Computer Science Technical Report CS-87-11, Brown University, June 1987.

[Hoare] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10, October 1974.

[Kepecs] Kepecs, J., "Lightweight Process for UNIX/Implementation and Applications," *USENIX Association Summer Conference Proceedings*, June 1985.

[Redell] Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C., "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, Vol. 23, No. 2, February 1980.

[Stroustrup] Stroustrup, B., "A Set of C++ Classes for Co-routine Style Programming", AT&T Bell Laboratories Computer Science Technical Report, Available with Release notes for 1.2.1 C++.

[Tevanian] Tevanian, A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W., "Mach Threads and the UNIX Kernel: The Battle for Control," *USENIX Association Summer Conference Proceedings*, June 1987.

SUMMARY

The Design of a Multiprocessor Operating System

Roy Campbell, Vincent Russo and Gary Johnston

University of Illinois at Urbana—Champaign
Department of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801-2987

Evolving applications and hardware are creating new requirements for operating systems. Real-time systems, parallel processing, and new programming paradigms require large adaptive maintenance efforts to modernize existing operating systems. An alternative to such adaptive maintenance is to seek new operating system designs that exploit modern software engineering techniques and methodologies to build appropriately structured modular software. This paper describes an approach to constructing operating systems based on a class hierarchy and object-oriented design and discusses the benefits and difficulties of realizing this design using C++.

Choices, a *Class Hierarchical Open Interface for Custom Embedded Systems*, is designed as an object-oriented system. Applications access the system through an object-oriented interface. Choices is being built by the Embedded Operating System (EOS) project at the University of Illinois at Urbana-Champaign. The architecture embodies the notion of a *customized* operating system that is tailored to particular hardware configurations and to particular applications. Within one large computing system containing many processors, many different specialized operating systems may be integrated to form a general purpose computing environment.

Choices is intended to exploit very large multi-processors interconnected by shared memory or high-speed networks. It uses a class hierarchy and inheritance to represent the notion of a family of operating systems and to allow the proper abstraction for deriving and building new instances of a Choices system. At the basis of the class hierarchy are multiprocessing and communication objects that unite diverse specialized instances of the operating system in particular computing environments. A set of software classes are provided that may be used to build specialized software components for particular applications including applications where high-performance is essential like data reduction or real-time control.

Of particular concern during the development of the system is whether or not the class hierarchical approach can support the construction of *entire* operating systems. C++ was chosen as an implementation language because it supports class hierarchies and inheritence while imposing negligible performance overhead at run-time. A software monitor is being used to evaluate the performance of Choices on an Encore Multimax with DPC processors. Although it is difficult to provide a meaningful performance measurement of an operating system, we have obtained results that are encouraging. System call overhead (including a trap and change to supervisor state) is approximately 38 microseconds which compares favorably with UNIX (approximately 170 microseconds). This is only about four times the overhead for a normal procedure call. Process creation time is approximately 3.8 milliseconds but includes creation of new virtual memory "spaces" for the process. The current process switching time is 536 microseconds. Further tuning will improve these figures. The paper describes the class hierarchy, call graph, and performance of the Choices kernel for the Multimax.

[1] This work was supported in part by NASA under grant no. NSG1471 and by AT&T METRONET.

Extending Stream I/O to Include Formats

Mark Rafter
mcvax!warwick!rafter
Computer Science Department
Warwick University
England

ABSTRACT

The C++ stream I/O system can be extended to allow formatted I/O in the style of stdio. This extension may be layered on top of the stream I/O system, requiring only minor changes to *<stream.h>*. The key traits of the original stream I/O system, namely extensibility and type-security are retained. An example of its use is:

```
int    i;
cin[ "%o" ] >> i;
cout[ "log of %d is %7f" ] << i << log(i);
```

The method used in the construction of the formatted I/O system takes it simplest form in a completely object-oriented language, i.e. one in which there is a type *obj* from which all other types are derived. C++ is not such a language, but its inheritance mechanism together with user-extensible type conversions allow the same approach to be used. For the moment we pretend that C++ is completely object oriented; the main points of the scheme are:

The type *obj* is equipped with virtual *read* and *print* functions. These are redefined to provide formatted I/O on a type by type basis.

Index operators are defined that bind together an unformatted stream and a format-specifier to make a formatted stream (types *fostream* or *fistream*).

fostream is provided with a *single* output operator (input is similar). This operator outputs objects of type *obj*, and hence may be applied to all objects. At the appropriate point, the output operator calls the *print* virtual function of its right-hand operand.

The idea is to provide a control-flow framework; the virtual functions handle the type-specific formatting work. To adapt this scheme to C++ we must simulate a unified type hierarchy. To do this:

Introduce a base class, *obj*, equipped with declarations for *read* and *print* virtual functions.

Inject each C++ type, *T*, into the *obj* type hierarchy, by deriving a class *obj_T* from *obj*. In formatted I/O expressions, *obj_T* objects act as containers that "wrap up" *T* objects.

Ensure that each of the container classes have a constructor that will automate the "wrapping up" process.

This sounds a little complicated but most of the work is scaffolding that is in the formatted I/O library. To provide a data-type with formatted I/O costs the user six lines of code, in addition to the data-type specific *read* and *print* functions. This additional code is very simple, and would be unnecessary if parameterised type-hierarchies were available; alternatively, its production can be automated with a suitable cpp macro.

What is "Object-Oriented Programming"?

Bjarne Stroustrup

ABSTRACT

"Object-Oriented Programming" and "Data Abstraction" have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula67, and Smalltalk. The general idea is to equate "support for data abstraction" with the ability to define and use new types and equate "support for object-oriented programming" with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

Introduction

Not all programming languages can be "object oriented". Yet claims have been made to the effect that APL, Ada, Clu, C++, LOOPS, and Smalltalk are object-oriented programming languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and CHILL. Could there somewhere be proponents of object-oriented Fortran and Cobol programming? I think there must be. "Object-oriented" has in many circles become a high-tech synonym for "good."

This paper presents one view of what "object oriented" ought to mean in the context of a general purpose programming language.

§2 Distinguishes "object-oriented programming" and "data abstraction" from each other and from other styles of programming and presents the mechanisms that are essential for supporting the various styles of programming.
§3 Presents features needed to make data abstraction effective.
§4 Discusses facilities needed to support object-oriented programming.
§5 Presents some limits imposed on data abstraction and object-oriented programming by traditional hardware architectures and operating systems.

Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

> *Decide which procedures you want;*
> *use the best algorithms you can find.*

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions.

Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in the program size. A set of related procedures with the data they manipulate is often called a *module* . The programming paradigm becomes:

> *Decide which modules you want;*
> *partition the program so that data is hidden in modules.*

This paradigm is also known as the "data hiding principle". Where there is no grouping of procedures with related data the procedural programming style suffices.

Programming with modules leads to the centralization of all data of a type under the control of a type manager module.

Data Abstraction

Languages such as Ada, Clu, and C++ allows a user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*†. The programming paradigm becomes:

> *Decide which types you want;*
> *provide a full set of operations for each type.*

Where there is no need for more that one object of a type the data hiding programming style using modules suffices.

Object-Oriented Programming

The problem with Data abstraction is that there is no way of expressing a distinction between the general properties of a concept (such as a geometric shape) and the properties of a particular variation the general concept (such as a circle or a square). Expressing such distinctions and taking advantage of them defines object-oriented programming. The programming paradigm is:

> *Decide which classes you want;*
> *provide a full set of operations for each class;*
> *make commonality explicit by using inheritance.*

Where there is no such commonality data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to an application area.

---

† I prefer the term "user-defined type": "*Those types are not "abstract"; they are as real as* int *and* float."
Doug McIlroy. An alternative definition of *abstract data types* would require a mathematical "abstract" specification of all types (both built-in and user-defined). What is referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities.

A C++ Object-Oriented Program Support Class Library

Keith E. Gorlen
Division of Computer Research and Technology
National Institutes of Health
Bethesda, MD 20892

INTRODUCTION

The Object-Oriented Program Support (OOPS) class library[1] is a collection of C++ classes similar to the fundamental (non-graphical) classes of Smalltalk-80. The OOPS library includes generally useful data types such as *String*, *Date*, and *Time*, and most of the Smalltalk-80 container classes such as *OrderedCltn* (indexed arrays), *SortedCltn* (sorted indexed arrays), *LinkedList* (singly-linked lists), *Set* (hash tables), and *Dictionary* (associative arrays). Classes *Process*, *Scheduler*, *Semaphore*, and *SharedQueue* provide support for multiprogramming with coroutines. One of the most powerful features of the library is the object I/O facility, which can convert arbitrarily complex data structures comprised of OOPS and user-defined objects into a machine-indepenent form that can be saved on disk files or moved between processes.

Initially, writing the OOPS library was an exercise for learning C++, a test of the quality of the C++ translator, and an experiment to see if C++ would support object-oriented programming (O-OP) in the Smalltalk-80 style. Since then it has become useful as a test suite for C++/C compilers, as a framework for O-OP in C++, for fast prototyping, and for a few production systems.

OVERVIEW OF THE OOPS CLASS LIBRARY

Class *Object*

Class *Object* is the most general OOPS class. All other OOPS classes, and classes written by a user to extend OOPS for a particular application, are ultimately derived from *Object* and inherit member functions that compare, copy, print, read, store, and retrieve objects. Other inherited functions test the class of an object and perform error handling.

Class *Object* completely implements some of these functions so they work for objects of any class, including user-written classes, without requiring additional code. However, many of them are virtual functions that derived classes must implement in order to work. Class *Object* implements these to issue an error message if they are applied to objects of a class that does not provide its own implementation. To help programmers properly write these functions for their own OOPS classes, we have developed template files to serve as a guide.

Class *Class*

Class *Object* declares the virtual function *isA()*, which returns a pointer to a description of the object to which it is applied. This description is itself an object, an instance of the OOPS class *Class*, and it contains information such as the name of the object's class, the size of the object, and a pointer to the *Class* object that describes instances of the object's base class. There is one instance of *Class* for each OOPS class linked with a program. These are inserted in a *Dictionary* object called the *classDictionary* so that the description for a class can be found given the name of the class.

Object I/O

The virtual member function *storeOn()* converts an object into a stream of ASCII characters that can be saved on a disk file or sent to another process. *StoreOn()* calls itself recursively to handle complex objects that contain member objects or pointers to objects. Since any pointers to objects are not valid in the context of another process, *storeOn()* converts these to object numbers, and it also converts multiple pointers to an object to the same object number.

The function *readFrom()* reads a stream produced by *storeOn()* and reconstructs the original object. *ReadFrom()* uses the *classDictionary* to find the descriptions for the class names it reads, and calls itself recursively to handle complex objects. It converts the object numbers assigned by *storeOn()* into pointers to the reconstructed objects, preserving the structure of the original object.

Both *storeOn()* and *readFrom()* require the support of (usually simple) member functions specifically written to handle the member variables of each class.

CONCLUSION

The OOPS class library demonstrates that C++ can support a class library with much of the basic functionality of Smalltalk-80. The advantages are that it is efficient and compatible with the UNIX/C environment. Some important characteristics of Smalltalk-80 are not implemented; for example, fundamental data types (like integers and characters) and blocks of code are not objects, and there is no automatic garbage collection. The lack of automatic garbage collection is a source of serious programming errors, but adding this feature to C++ would make it too inefficient for many applications.

REFERENCE

1. K. E. Gorlen, 'An Object-Oriented Class Library for C++ Programs', *Software – Practice and Experience* (in press).

An Object-Oriented Class Library for C++

Ken Fuhrman
Ampex Corporation

Introduction & Goals

An Object-Oriented Class Library for C++ was developed for a project at Ampex. It runs on Unix systems as well as with an internally developed Ampex Real-Time OS (ARTOS). This class library provides an extensible method for designing objects based around the basic class hierarchy provided in the library.

The following goals were identified when designing the Object Class Library:

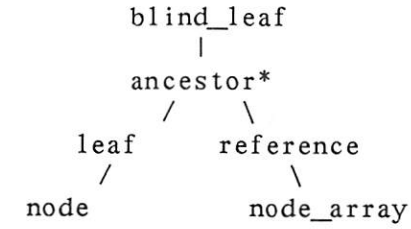
- Provide support for persistent objects.
- Provide support for constructing dynamic objects.
- Allow the design of arbitrarily complex objects.
- Optimize for size and speed in traversal of dynamic objects.
- Reduce code redundancy.
- Develop a consistent methodology for object design.
- Provide a basic set of operations for the construction of dynamic objects.
- Support an extensible method for further additions to the library.

If all objects are created from the basic classes, then a consistent methodology and formulation of objects can be provided. The Basic Object-Oriented Class Library does not provide any specific objects as such. It provides a set of basic classes that allow the creation of more specific objects by an application programmer. Certain useful objects can then be built into the Object-Oriented Library as desired. This provides a method to extend the capabilities of the library.

Overview of the Basic Object-Oriented Class Library

The Object-Oriented Class Library is designed around five basic classes that are organized into a class hierarchy. A full set of operations are provided that allow the construction, traversal and editing of objects that have been derived from one of the 5 basic classes. There are two ways to create objects. The first is to derive simple objects directly from one of the basic classes. The second way is to construct dynamic objects from other simple or dynamic objects. The concept of construction is integral to the Object-Oriented Class Library and many of the basic set of operations are provided for the construction of more complex objects.

The 5 basic classes are: blind_leaf, leaf, reference, node and node_array. The class hierarchy is shown below:



*ancestor cannot be derived from

All objects derived from the basic classes are persistent and self- decoding in terms of size and type. The size of an object is determined at run-time thus variable sized objects can be persistent. As objects move down the hierarchy, based on the class they are derived from, more capability is provided. Currently virtual functions are not used, although some of the added capabilities for printing & debugging will use virtual functions.

Some of the capabilities of the Object-Oriented Class Library are the construction of dynamic objects, persistent objects, built in exception handling, properties, and object references. The

Object-Oriented Class Library is suited for creating dynamic objects that are list or tree based, although more complex data structures can be constructed.

Object Construction

Construction allows more complex objects to be dynamically built from other objects. Normally a programmer would design an object that has both data and operations associated with it. This object is then derived from one of the basic classes. The operations for the object would use the basic operations provided as part of the library. The programmer can use the basic construction operations or write more specific operations for constructing objects.

Properties

Properties are attached to an object on the property list. There are a set of operations that allow editing and traversal of the property list. No interpretation is done on the property list by the basic operations, this is left up to the object itself. An example of properties might be transform matrices that are attached to the property list of a graphical primitive object.

References

References allow two separate objects to share (reference) the same object. Normally a reference is not persistent and when an attempt is made to save an object that contains a reference an error occurs. There is the concept of a library reference that allows the reference to exist in files. The library understands the organization of how objects are saved to files and can reference an object that is stored in a file. The library is an object database.

Basic Operations provided in the Object-Oriented Class Library

The basic operations are divided into the following groups, List Operations, General Operations, Reference Operations, Operations on Substructure, Operations on Properties, and Diagnostic Operations. The Object-Oriented Class Library interface specification has all of the details on the basic set of operations that are provided.

Future Directions

The concept behind the Basic Object-Oriented Class Library is to provide a methodology for the creation of more complex objects. These objects have specific applications such as a dictionary or semaphore. Some objects could have a universal appeal, such as OS interfaces, Graphics Interfaces, or general Database Applications. The goal is to provide a standard object-oriented approach to data structure design and allow the library to be easily extended. In the future it is desired to implement more I/O capability for printing the data structures as well as providing builtin debugging capability.

Teaching C++

Tsvi Bar-David

Abstract

The purpose of this talk is to share my experiences gained teaching C++ and to suggest how the language is best taught. My experience is based on having taught most of the C++ classes offered at AT&T Bell Laboratories over the past year.

Here are the observations:

- The students typically have a fine understanding of the C language expression syntax. This helps them learn C++ quickly since the language is (by and large) derived from C.
- The student population, although technically sophisticated, typically have a weak grasp of C language scoping mechanisms. This deficiency makes it more difficult to teach the extended scoping rules of C++, including object data within the scope of member functions, inheritance and virtual functions.
- Students usually come to the C++ class with little or no knowledge of object oriented programming. This is a dual problem. First, the language is intended for making it easy to design and implement using the object paradigm. Second, existing documentation on C++ explain well the nuts and bolts of the language, however they do not address *how* to do object oriented programming in the language. The result is that students program in a C-like way in C++, losing much of the power of the language.

Here are some conclusions:

- Teach students object oriented design and implementation as a pre-requisite to learning C++ syntax. I have already piloted such a 1-day course with good results.
- Ensure that C++ students have a *solid* grounding in the details of the C language.

Modelling Graphical Data with C++

Al Conrad

University of California
at Santa Cruz

ABSTRACT

Some features of object oriented programming make it particularly well suited to applications in graphics modelling. Polymorphism obviates the need for the countless case statements usually found within graphics operations like draw or resize. The operator overloading mechanism of C++ allows the programmer to use a reasonable notation when coding operations like: "add shape to picture" or "delete all shapes within a given rectangle". This paper suggests an approach for taking advantage of these new opportunities afforded to the graphics programmer by C++ and examines the specific case of Tinker. Pf(SRTinker is a drawing program written for X windows by a UCSC undergraduate under the supervision of the author.

Abstract:
Using C++ to Develop a WYSIWYG Hypertext Toolkit

Jim Waldo
Apollo Computer Inc.
330 Billerica Rd.
Chelmsford, Ma. 01824
decvax!apollo!waldo

This paper is essentially a case study in using C++ to develop a toolkit for text applications. I will begin by giving an overview of the toolkit itself, focusing on the ways in which it differs from standard methods of dealing with text. I will then discuss how the toolkit utilizes various features of the C++ language, and how certain features of the language allow the actual implementation of the toolkit to more closely resemble the optimal system envisioned during the design process. I will conclude by discussing some of the dark linings of the silver cloud of C++ and discuss some of the methods we are using to attempt to overcome these problems.

The toolkit we are developing, known internally as the Text Management Library (TML), is meant to provide all the functionality needed to produce a broad range of text applications, from viewers to simple plaintext editors to structured document or program editors. The requirements specified that it should handle multiple fonts and character sets (both one and multiple byte per character), allow various forms of formatting control, support hypertext facilities for multiple paths through a document, and allow the presentation and manipulation of such text in a true WYSIWYG fashion at interactive speeds. To accomplish this, the library is based on a model that treats text as a hierarchically structured set of objects, the terminal nodes of which point off to a byte heap containing the characters that make up the text. These objects may have associated with them a set of properties that control the interpretation of the characters and the formatting and structural characteristics of the text. To present these objects visually (either on a screen or on a printer) the object is translated by a set of routines known as the formatting component into an abstract two-dimensional representation. This two-dimensional representation is in turn handed to the rendering component of the library, which does the actual drawing.

The library was originally implemented using Pascal as its base language. However, it was decided to redesign and reimplement the toolkit in C++ to take advantage of the portability and object oriented features of that language.

The implementation makes use of nearly all of the features of C++. Parallel class hierarchies are used in the modeling, formatting, and rendering components, allowing us to give the same "feel" to the components. The use of class hierarchies has also allowed us to introduce mechanisms for users to extend the set of objects that can be dealt with by the library, allowing the introduction of graphic objects of various kinds. By using both the virtual function and function overloading features of C++ we have been able to implement the notion of object traits. Traits are sets of semantically similar operations that may be performed on all the members of similar objects classes. This has led to a clear set of semantic relationships that unify the structures in the library. Operator overloading is used for the objects that make up the text model, allowing the usual lattice relations to be defined in a natural way on that structure. The power of constructors and destructors has been utilized to implement a generalized object workspace for text objects. In short, the matching of language and application has been nearly perfect.

The imperfections in the matching occur for the most part in the realm of saving text objects to disk. These problems revolve around the use of virtual functions in the classes of objects that are saved; the problem is to come up with a way of restoring these virtual functions when the objects are brought off of

disk to be used again. This is a minor (but non-trivial) problem when restricted to objects that are part of the basic ontological store of TML; the problem gets far more difficult when one introduces the ability to extend the set of objects to an application using the toolkit. I will conclude the talk with some of our current thoughts about how to deal with this problem.

The Design and Implementation of InterViews

(Presentation Summary)

Mark A. Linton and Paul R. Calder
Stanford University

InterViews is an object-oriented user interface package written in C++. The current implementation runs as a library on top of the X window system. The InterViews base class **interactor** and subclass **scene** define how interactive objects can be combined to form composite user interfaces. An interactor typically represents the user interface to some abstract data. We call this kind of interactor a **view** and the abstract data a **subject**. The separation of subject and view allows a user interface to be customized through the use of different views for the same subject. The name InterViews comes from the concept of interactive views.

All user interface objects are derived from the interactor class. Every interactor has a **shape** that defines its natural size, shrinkability, and stretchability. This information is used to allocate display space for an interactor, and the interactor's **canvas** is set to the actual space obtained. An interactor can perform output to its canvas using a **painter**. A painter provides drawing operations and manages graphics state such as color, font, fill pattern, translation, rotation, and scaling. An interactor can receive input events in one of two ways. It can either read input directly, filtered by a **sensor**, or it can be passed an event from another interactor.

A scene is an interactor that contains one or more other interactors. The **hbox** and **vbox** subclasses arrange component interactors side-by-side horizontally and vertically, respectively. Boxes shrink or stretch the interactors they contain to fit into available space. Within a box, **glue** interactors are used to define variable space between other interactors. Thus, a user interface defined in terms of boxes and glue can take on a variety of shapes and sizes without modification to component interactors.

The InterViews library currently includes **menu**, **button**, and **scroller** classes. A menu is represented as a box of **menu items**, where each item is itself an interactor. A button is a view of a **button state**; pressing a button sets the state to a particular value. Several buttons can be visible for the same button state. InterViews provides classes for push-buttons, radio-buttons, and check-boxes. Horizontal and vertical scrollers are views of an interactor's **perspective**. The perspective defines what portion of a subject is currently displayed by an interactor. A scroller displays a scroll bar reflecting the current perspective. The perspective can be modified by the user through the scroller, or by the interactor itself.

InterViews currently runs on MicroVAX and Sun workstations under X (either version 10 or 11). The InterViews library is roughly 11,000 lines of C++ source code. We have also implemented several applications on top of the library, including a reminder dialog box, a variable-size digital clock, a drawing editor, a load monitor, a window manager, and a display of incoming mail. We are currently working on a more general drawing system, a structure editor, and a visual debugger as part of a C++ programming environment. InterViews is available on the X11 distribution or via anonymous ftp from lurch.stanford.edu.

Research supported by the SUNDEC project through a gift from Digital Equipment Corporation.

The Design of the Allegro Programming Environment

(Presentation Summary)

Mark A. Linton, Russell W. Quong, and Paul R. Calder
Stanford University

Programmer productivity is a function of the time it takes to produce a desired software system. Software development is often *evolutionary*; that is, systems are gradually enhanced and debugged. The Allegro programming environment supports the evolution of software by representing program dependencies explicitly and by providing fast turnaround after a change. Allegro is aimed at evolution of production software systems, which are usually large, distributed across several machines, written in several languages, and compiled for run-time efficiency.

Allegro manages program information using a distributed object-oriented architecture. By "object-oriented" we mean that all program information is represented in terms of objects. By "distributed" we mean that objects can transparently reference and perform operations on objects located on remote machines. Because objects are too small and plentiful to be implemented as distinct files or processes, we group a number of related objects into an **object space**. An object space is like an address space with objects taking the place of bytes or words.

Objects in Allegro are organized into object spaces according to two criteria: (1) Objects with many interrelationships that as a whole represent a more complex object are usually stored in the same object space. For example, objects representing the source code for a program, including procedures, statements, expressions, and types are generally in the same space. (2) Object spaces are organized hierarchically just as files are in a hierarchical file system. For example, "/usr/linton/xyz/src" might refer to the object space containing the source objects for the program "xyz".

Program source is represented in an object-oriented intermediate language called SLIC (Source Level Intermediate Code). SLIC nodes are either **definitions**, **uses**, **sequences**, **forms**, or **units**. Every use is linked to its definition, and every definition is linked to all of its uses. Each definition is associated with a form that defines the components of instances, how to display and manipulate instances, and how to generate code for a compiler (implies closure for one-pass compiling).

Measurements of current program development practice indicate that most changes involve one or two modules and that modified modules usually do not change in size much if at all. Thus, incremental linking techniques can be used to produce a new executable quickly after a change. The basic approach is to associate with every symbol a list of addresses to relocate when the symbol's address changes. Also, modules usually can be updated in place in the executable if a small amount of extra "slop" space is allocated for each module.

We are in the process of making Allegro a usable environment for our own C++ program development. We have implemented an incremental linker that is more than an order of magnitude faster than the Unix linker when one module is changed in a large program. We have also implemented a library for performing remote object communication. We are working on a C++-oriented editor, a symbol manager, a process manager, and the debugging user interface.

Research supported by the SUNDEC project through a gift from Digital Equipment Corporation.

A C++ Class Browser

Raghunath Raghavan, Niranjan Ramakrishnan & Sue Strater

Mentor Graphics Corporation
8500 SW Creekside Place
Beaverton, OR 97005

Presentation Summary

Objectives

- o Selectively view class declarations in one or more header files
- o Quick response
- o Viewing of all visible member functions/variables of a class, including those defined in base classes
- o Viewing of code/comments related to member function/variable
- o Graphical display of class hierarchy

Strategies

- o Assume header files are self-contained
- o C pre-processor used to manage included files
- o Files can be in any directory
- o No database of classes / files
- o Parsing done only on demand

Capabilities

- o User specifies file(s) of interest. Browser generates list of class declarations found. Browser also generates tree of class derivations
- o User selects class to be parsed either from tree or from list Functions and variables of selected class displayed, including visible members from base class(es)
- o User selects member function/variable. Browser shows actual declaration, including comments, found in header file.
- o Search capabilities
- o Cut and paste

User Interface

- o Screenshots of browser
- o Explanation of individual windows
- o User's interaction with browser

Implementation

- o Description of underlying library components
- o Description of classes used to implement browser
- o Display - data structure correspondence

Pi: A Case Study in Object-Oriented Programming

T.A. Cargill

ABSTRACT

Pi is a debugger written in C++. This paper explains how object-oriented programming in C++ has influenced Pi's evolution. The motivation for object-oriented programming was to experiment with a browser-like graphical user interface. The first unforeseen benefit was in the symbol table: lazy construction of an abstract syntax-based tree gave a clean interface to the remainder of Pi, with an efficient and robust implementation. Next, though not in the original design, Pi was easily modified to control multiple processes simultaneously. Finally, Pi was extended to control processes executing across multiple heterogeneous target processors.

USENIX Association
P.O. Box 2299
Berkeley, CA 94710